Trabajo Práctico №3

Heurísticas

Junio 2017 Optimización

Integrante	LU	Correo electrónico
Hurovich, Gustavo Martín	426/13	gushurovich@gmail.com
López, Agustina Florencia	120/13	agustina.lopez@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria – Pabellón I, Planta Baja
Intendente Güiraldes 2160 – C1428EGA
Ciudad Autónoma de Buenos Aires, Rep. Argentina
Tel/Fax: +54 11 4576 3359
http://exactas.uba.ar

Introducción

A lo largo del curso hemos estudiado diferentes métodos para hallar óptimos de problemas de minimización con o sin restricciones. La mayoría de ellos asumen algún buen comportamiento de las funciones a optimizar, por ejemplo que éstas son contínuas o incluso diferenciables.

Sabemos que, en la realidad, esto no es siempre así. Es frecuente toparse con problemas concretos en los que nuestra función objetivo no es lo suficientemente buena para usar las técnicas que aprendimos. Es por eso que, en este trabajo, se desarrollarán diferentes métodos heurísticos que no tendrán la necesidad de afrontar estos inconvenientes.

En base a la experiencia, se considera que las heurísticas que analizaremos son útiles y eficientes, pudiendo conseguir óptimos reales sin necesidad de hacer grandes asunciones sobre las funciones a minimizar. En particular, desarrollaremos 4 métodos heurísticos:

1. Descenso aleatorio
   a) Descenso más rápido
   b) Búsqueda local
   c) Búsqueda local iterada
2. Algoritmo genético
3. Recocido simulado
4. Algoritmo a elección: Grandes mejoras para recocido simulado

Todos los algoritmos implementados recibirán parámetros similares. La función f a minimizar, un dominio rectangular definido por límites inferiores y superiores en cada coordenada, una cantidad máxima N de iteraciones del algoritmo, y una variable en $\{0, 1, 2\}$ que indica si hay que graficar el error a lo largo de las iteraciones, los puntos recorridos del dominio, o nada (1, 2 y 0 respectivamente). Por supuesto, cada uno tendrá sus particularidades, por lo que requerirá de más parámetros.

1. Descenso aleatorio

1.1. Descenso más rápido (A11)

1.1.1. Desarrollo

Este método heurístico es bastante simple, y podríamos considerar que su base es un poco de "*fuerza bruta*". La idea general consiste en, dada una función a minimizar f y un intervalo donde decidamos buscar su óptimo, elegir un punto de comienzo dentro de dicho intervalo, y luego generar otros puntos al azar (que nunca se escapen de nuestra región) verificando en cada paso si son mejores que el que ya teníamos y guardándolos en tal caso. Es decir, estamos tanteando a ciegas en todo el dominio, quedándonos con el mejor de todos los puntos visitados. Aquí, dado que queremos encontrar un mínimo de f, consideramos que x *es mejor* que x_0 si $f(x) < f(x_0)$.

1.1.2. Implementación

La implementación de este algoritmo fue muy sencilla. El usuario deberá proporcionar la función f a minimizar, un valor mínimo y un valor máximo que pueda tomar esta función. Además, será también quien elija la cantidad N de iteraciones que el algoritmo haga. Claro está que mientras más grande sea el número de iteraciones elegido, mejor será la aproximación al mínimo que tendrá este método (sujeto a no exceder un cierto tiempo de corrida deseado). Este programa toma como punto de inicio el valor mínimo de f proporcionado, y lo considera el mejor hasta el momento. Luego, en cada uno de los N pasos, genera un punto al azar dentro de la región de búsqueda elegida anteriormente. En caso de que el valor de f en dicho nuevo punto sea menor al obtenido anteriormente, actualizamos nuestro mejor valor como el último encontrado. Finalmente, al terminar todas las iteraciones, devolvemos el punto que haya alcanzado el valor más pequeño de nuestra función.

Una implementación alternativa podría haber sido elegir N puntos al azar en el dominio, evaluar la función en todos ellos, y tomar mínimo en ese arreglo.

1.2. Búsqueda local (A12)

1.2.1. Desarrollo

Este algoritmo es un poco similar al anterior, pero tiene una diferencia sustancial: ya no probamos mejorar nuestra función tomando puntos al azar en cualquier parte de nuestra región, si no que lo hacemos en un entorno del mejor punto que tenemos hasta el momento.

1.2.2. Implementación

Primero que nada, más adelante veremos que vale la pena pedir que nuestro programa devuelva dos valores: el punto donde consideramos que se alcanza el óptimo y su valor de f. También, será útil que reciba un punto inicial en vez de considerar a discreción alguno en el espacio.

El algortimo sigue siendo sencillo. Dados f, un valor máximo y mínimo, una cantidad N de iteraciones, un punto inicial x_0 y un valor arbitrariamente chico ϵ, guardamos a x_0 como nuestro mejor valor hasta el momento (y por supuesto también guardamos $f(x_0)$). Consideramos dos nuevos límites para nuestro espacio: el máximo entre el mínimo del intervalo proporcionado y $x_0 - \epsilon$, y el mínimo entre el máximo del intervalo y $x_0 + \epsilon$ (es decir, intersecamos la bola centrada en x_0 de radio ϵ con el dominio de la función).

Iteramos como antes, actualizando x_0 y $f(x_0)$ cada vez que encontremos una mejora. Para esto, tomamos puntos al azar pero que ahora caigan dentro de estos nuevos valores recientemente armados, garantizando así que los nuevos puntos nunca se escapen de la región donde buscamos optimizar f pero siempre caigan dentro de un entorno *cuadrado* de lado ϵ del mejor punto que llevamos encontrado. Es decir, para cada elemento de mejora que vamos seleccionando, se realiza una *búsqueda local* con el objetivo de seguir minimizando nuestra función.

Es importante recalcar que, a valores muy pequeños de ϵ, la sucesión puede quedar estancada en un mínimo local. Aunque por otro lado, para valores grandes, termina siendo igual que el primer método mencionado.

1.3. Búsqueda local iterada (A13)

1.3.1. Desarrollo

La idea general de este algoritmo es combinar los dos anteriores: realizar búsquedas locales en diferentes lugares del espacio, elegidos al azar. Es un poco más sofisticado que los anteriores pero tal vez más adecuado, sobre todo porque el riesgo de caer en un mínimo local ahora debería ser menor.

1.3.2. Implementación

El usuario brindará la función a minimizar, los extremos del espacio deseado, la cantidad N de iteraciones y, como antes, un valor chico ϵ. Por tomar un punto inicial como el mejor actual, vamos a considerar el centro del intervalo brindado. En este caso, en realidad el valor N nos indica la cantidad de reinicios al azar que haremos en la región indicada. Tomamos un punto al azar de todo nuestro espacio. Llamamos entonces a la función *búsqueda local* explicada anteriormente para encontrar un buen "óptimo" en el entorno de este punto, con $N = 1000$. Si la función f en este nuevo elemento toma un valor menor al actual, entonces reemplazamos por éste a aquel punto que considerabamos mejor (aquí vemos la utilidad de que la función recién mencionada tome un valor inicial como parámetro y devuelva el punto óptimo y su valor de f). Repetimos este proceso recorriendo *pequeñas* regiones de todo el espacio y quedándonos entonces con el mejor punto encontrado.

2. Algoritmo genético (A2)

2.1. Desarrollo

Los algoritmos genéticos se inspiran en la evolución biológica e intentan llegar a una solución al problema haciendo una analogía con lo que conocemos en biología como *selección natural*. A partir de una *población inicial* (soluciones), la idea será quedarnos con algunos *individuos* (los más aptos) para luego generar su descendencia mediante dos procesos: *crossover* y *mutación*. Esto da lugar a una nueva generación de hijos que puede o bien incorporarse a una parte de la anterior, o bien reemplazarla completamente. En cualquier caso, se reinicia el proceso para dar lugar a más generaciones y eventualmente encontrar el mejor *individuo* (o los mejores).

2.2. Implementación

Nos topamos ahora ante un algoritmo un poco más complicado de implementar, pero sobre todo ante un proceso más complicado a la hora de tomar decisiones: quiénes y cuántos son los mejores individuos, cómo creamos a sus hijos, si éstos se incorporan a la generación o los reemplazan, cómo realizamos el crossover y la mutación, etc.

Una de las primeras elecciones que hicimos fue quedarnos con la mejor mitad de la población inicial (donde *mejor* nuevamente significa tener un valor menor para f). Luego, decidimos formar parejas entre ellos y que cada una genere dos hijos. Para que esto sea correcto, necesitamos que el tamaño de la población

inicial sea un número múltiplo de 4 (al quedarnos con la mitad de esa generación, nos queda un tamaño que es múltiplo de 2 y que puede ser subdividido en parejas de forma precisa). Dado que la cantidad de individuos de la población inicial (*popsize*) es un parámetro de entrada de nuestro programa, modificamos este número agregando algunos individuos hasta obtener el múltiplo de 4 más cercano, mientras que sea mayor. Consideramos que esto no afecta el análisis requerido por el usuario ya que se espera que *popsize* sea un número grande y a lo sumo lo estamos incrementando en 3 unidades.

Hecho esto, inicializamos nuestra población generando la cantidad mencionada de soluciones (de dimensión n) equiespaciadas entre el valor mínimo y el máximo que pueda tomar f, parámetro brindado por el usuario. Las almacenamos en una matriz de tamaño *popsize* x $n+1$, en cuya última columna guardamos, para mayor comodidad, el valor de la función a analizar en esta generación. Luego, utilizamos el comando *sortrows* de Matlab para ordenar todas las filas de la matriz según los valores de la última columna, de menor a mayor. Esto nos asegura que tenemos a nuestra población ordenada según su "*aptitud*": los *mejores* individuos nos quedan en las primeras filas de la matriz. Esto nos permite fácilmente seleccionar la mitad de toda la generación, que son aquellas criaturas que vamos a utilizar para armar parejas.

El paso siguiente es justamente lo que acabamos de mencionar: formar las parejas. Después de considerar varias opciones posibles, decidimos ir tomando pares entre los mejores y los peores individuos antes seleccionados. Por ejemplo, supongamos $popsize = 100$. Una vez ordenada la población en torno a su aptitud y luego elegidos los mejores 50, formamos las parejas $1-50, 2-49, 3-48$, etc.

Finalizado esto, entrecruzamos cada dúo para que den lugar a dos hijos (*crossover*). Creadas ya nuestras nuevas soluciones (notar que tenemos tantos hijos como padres, en nuestro ejemplo: 50 hijos), decidimos *mutarlas* para que luego reemplacen por completo a la generación de padres que no usamos, los "malos" (entrarían en lugar de los individuos 51-100).

Así, conseguimos una nueva población que supone mejorar la original. Repetimos este proceso N veces, número determinado por el usuario, con la esperanza de finalmente conseguir una muy buena generación que pueda acercarnos a un mínimo.

La elección de las parejas "primero con último", surgió para evitar estancamiento. Cada vez que un hijo desplaza a uno (o varios) padres, las parejas se rearman, por lo que aparecen nuevos hijos. Si los hubiéramos elegido "de a pares" (el primero con el segundo, el tercero con el cuarto, etc.), es muy probable que a partir de un momento los primeros dos no se modifiquen, por lo que sus hijos serán siempre los mismos (módulo mutaciones).

Una posible mejora podría ser: a cada padre, asignarle un peso, de manera que los mejores tengan mayor peso que los peores. Luego, elegir las parejas al azar, eligiendo a los padres con una probabilidad proporcional a su peso. De esta manera, priorizamos la aparición de parejas entre "los mejores padres", pero evitando el estancamiento mencionado.

2.2.1. Crossover

Nuevamente nos vimos ante una oportunidad en la que teníamos que tomar una decisión. Esta vez acerca de cómo combinar a los padres para su reproducción. Luego de analizar varias alternativas, aprovechamos el hecho de tener que generar dos hijos para entrecruzarlos equitativamente.

Para el $hijo_1$ consideramos $\frac{2}{3}$ de la carga genética de uno de sus padres, y $\frac{1}{3}$ del otro. Es decir, tomamos una combinación convexa entre ambos puntos considerados, pesando más al primero. Para el caso del $hijo_2$, hacemos exactamente lo mismo pero invirtiendo los roles de los progenitores.

2.2.2. Mutación

Para esta variación genética, decidimos alterar cada individuo bajo un ajuste normal. Es decir, a cada solución numérica que habíamos obtenido como hijo, le sumamos una normal con esperanza nula y $\sigma = \frac{xmax-xmin}{20}$. Finalmente, nos garantizamos de que esta modificación genere nuevas soluciones que permanezcan en nuestro espacio original.

Hay muchas otras posibilidades para considerar al hacer mutaciones. Sin embargo, preferimos una que sea razonable para cualquier función de entrada, y no que aproveche las características de una, o de un grupo particular.

3. Recocido simulado (A3)

3.1. Desarrollo

Al igual que en el caso de los algoritmos genéticos, este método es sustancialmente diferente a los otros.

Primero que nada, vamos a tener presentes dos valores, el mejor hasta el momento, que llamaremos B y el valor actual con el que estamos trabajando, S. La idea será ir modificando S (bajo algún criterio a elección), y en cada paso, ver si la modificación hecha, que llamaremos R, fue para mejor (en este caso, si el valor de f en R es menor al de S). Si esto efectivamente pasó, entonces ese es nuestro nuevo valor actual que consideraremos; es decir, reemplazaremos a S por R. Si no, eventualmente lo aceptaremos pero con una cierta probabilidad:

$$p(t, R, S) = e^{\frac{f(S)-f(R)}{t}} \tag{1}$$

Notar que, si no aceptamos R en una primer instancia, es porque $f(R) > f(S)$. Esto nos dice que el exponente del valor que acabamos de plantear es negativo, con lo cual efectivamente se toman valores entre 0 y 1.

Ahora bien: ¿qué es t?. Este parámetro es conocido como "*temperatura*" y va disminuyendo a lo largo de las iteraciones. Notemos que mientras más chico es el valor de t, el exponente de la probabilidad que planteamos es más grande (y recordemos que era negativo), de donde toda esta exponencial toma valores cada vez más chicos. En criollo: disminuyendo la temperatura en cada paso, nos garantizamos que las chances de aceptar una solución que no mejora nuestra función objetivo sean cada vez menos.

Vale la pena tomar una temperatura inicial bastante alta, de modo que primero se acepten varias soluciones (que puedan o no ser mejores), para así obtener un recorrido general por nuestro espacio.

Cabe destacar que, al finalizar cada iteración, se actualiza el valor de B en caso de que R sea un mejor candidato a solución.

3.2. Implementación

Como vinimos haciendo en el resto de los algoritmos implementados, f, la cantidad N de iteraciones y los extremos máximos y mínimos donde buscar el óptimo de la función son parámetros a proporcionar por el usuario. Esta vez, incorporamos también T que representará a la temperatura inicial. En nuestro caso, $B = bestx, S = actualx, R = a$. Como mencionamos antes, tenemos la libertad de elegir la modificación que queramos hacerle a S para transformarlo en R. Nosotros decidimos sumarle un ruido gaussiano, de varianza fija igual a 1. Calcular la probabilidad deseada es cuestión de sólo hacer la cuenta, pero para que el programa acepte o rechace R en base a esta probabilidad, utilizamos el comando *rand* de Matlab, que nos otorga un número aleatorio entre 0 y 1 (de hecho, $rand \sim U[0, 1]$). Para reducir la temperatura, simplemente la multiplicamos por 0.99 en cada iteración.

4. Algoritmo a elección: versión mejorada para recocido simulado (A4)

4.1. Desarrollo

Como algoritmo libre decidimos hacer varios retoques a recocido simulado. Fuimos incorporando varias herramientas a lo largo de su desarrollo, garantizandonos que efectivamente se consiguieran buenos resultados.

En primer lugar, quisimos mejorar la forma de modificar S, es decir, cambiamos la forma de armar R. Ahora, le sumamos a S una normal con esperanza 0 y una varianza que se modifica en cada iteración. Si estamos en la iteración i (de las N totales), entonces $\sigma = \frac{10}{i}$. Esto nos permite perturbar S lo suficiente como para acercarnos a un óptimo, de manera que esta modificación sea cada vez menos notoria, ya que suponemos que a medida que avanzan las iteraciones, estamos más cerca de encontrar un mínimo.

Básicamente, no solo reducimos la temperatura (que reduce la chance de aceptar soluciones peores) sino que también reducimos el rango de búsqueda (bajando la chance de irnos lejos del punto actual).

En segundo lugar, decidimos no bajar la temperatura en cada paso, sino hacerlo cada diez iteraciones. De esta manera nos garantizamos tener un mejor recorrido global del espacio donde estamos optimizando nuestra función, pues la probabilidad con la que aceptamos un valor "*no mejor*" es bastante alta (cercana a 1) en los primeros pasos del algoritmo.

Por otro lado, al probar este algoritmo, notamos que muchas veces la localidad en la que quedamos no contiene al mínimo global. Por lo tanto, tiene sentido introducir reinicios (como hicimos en la búsqueda local iterada). Para ello, decidimos llevar la cuenta de cuántas iteraciones llevamos sin mejorar el óptimo global; si esta cantidad supera un umbral, significa que hace un tiempo ya que no estamos obteniendo mejoras. En ese caso, elegimos a como un punto al azar y volvemos a empezar.

Ahora bien, cuando realizamos un reinicio, también hay que reiniciar la temperatura (pues si no, empezamos en otro lado con temperatura muy baja) y la varianza de la búsqueda local.

En este punto, es importante notar que la motivación para utilizar recocido simulado es no estancarse en una vecindad, sino que permitir recorrer el espacio, hasta finalmente quedarse en una. Ahora bien, una vez que elegimos una localidad, ¿por qué no buscar efectivamente el óptimo de ella? Además, ¡ya tenemos una forma de hacerlo!

La última mejora consistió entonces, en realizar sobre el mejor punto encontrado por este algoritmo, una cantidad de iteraciones del Método del Gradiente. Este cambio resultó muy bueno, ya que se consiguen aproximaciones muy precisas al óptimo real, incluso en una dimensión grande para funciones poco amigables, como la de Rosenbrock.

La única pequeña desventaja que encontramos para nuestro algoritmo es que este último cambio, el del método del gradiente, no sería útil para alguna función que no sea C^1, por ejemplo (al menos en un entorno. Sin embargo, el valor mínimo encontrado antes de que comience este proceso final ya es suficientemente bueno, por lo que consideramos que no es un problema.

4.2. Experimentación y Resultados

4.2.1. Tiempos

En primer lugar, nos interesa saber si hay gran diferencia entre las variantes propuestas en cuanto a tiempos de corrida. Para ello, suponemos que no importa qué función decidamos testear, pues la cantidad de iteraciones es fija. Sin embargo, el permitir que las funciones tengan dominios de dimensiones distintas puede afectar a los tiempos, por lo que consideramos dos funciones distintas: una función cuadrática en $\mathbb{R}^2$, y la función de Rosenbrock en $\mathbb{R}^4$.

A continuación mostramos la tabla con los resultados obtenidos. Los tiempos están medidos en segundos, utilizando librerías de Matlab.

Cuadro 1: Tiempos de corrida en segundos

	Cuadrática	Rosen
Descenso más rápido	0.3625	0.5275
Búsqueda local	0.3234	0.4123
Búsqueda local iterada	0.5757	0.7799
Genético	35.7293	40.3100
Recocido Simulado	0.4301	0.6041
Variante de Recocido.	4.5615	5.8502

Todos los casos fueron corridos con N = 100000, y los parámetros acorde a los correspondientes algoritmos y funciones. El código fuente puede encontrarse en el archivo "tiempos.m". La única excepción fue la Búsqueda local iterada. Como en ella, para cada iteración realizamos N/10 iteraciones de búsqueda local, elegimos N=1000, de modo que $N \times N/10 = 100000$.

Como esperábamos, los primeros tuvieron un desempeño similar, siendo la búsqueda local iterada un poco peor. Esto tiene sentido, siendo que combina las dos anteriores. Por otro lado, también se cumple consistentemente que todos los algoritmos son más lentos con la función de Rosenbrock en dimensión 4.

El algoritmo genético armado resultó un par de órdenes de magnitud más lento que el resto. Es importante recalcar que fue corrido con un tamaño de población de 40. Por lo tanto, cada iteración realiza 20 crossovers, 20 mutaciones, y luego reordena un vector de 40 elementos. Por lo tanto, tiene sentido que ande 50 o 60 veces más lento que los anteriores.

Recocido simulado, en su versión original, es muy parecido a la búsqueda local iterada, y los tiempos se condicen con esto. Sin embargo, nuestra variante resulta más lenta, pues realiza más operaciones, llama al método del gradiente, etc. A continuación veremos que, a pesar de esto, resultó nuestra opción preferida, pues sus resultados superaron con creces a los del resto en cuanto a calidad. En ese sentido, si necesitáramos por algún motivo un algoritmo veloz, eligiríamos nuestra variante, con una cantidad menor de iteraciones.

4.2.2. Distancia al mínimo, a lo largo de las iteraciones

Para realizar este análisis, tomamos como función test a la de Rosenbrock en dimensión 7. Ésta tiene un mínimo global en $x^* = (1, 1, 1, 1, 1, 1, 1)$, con $f(x^*) = 0$. Por otro lado, tiene un mínimo local en

(**-1**, 1, 1, 1, 1, 1, 1). En varias ocasiones, resultó que alguno de los algoritmos cayó en ese punto. Sabemos que es algo inevitable, dada la naturaleza del problema, aunque por supuesto preferiríamos que esto no suceda.

Tanto A11 como A13 realizan constantes reinicios, por lo que el valor de la función objetivo oscilará al azar. En efecto, en ambos se obtiene algo similar a esto:

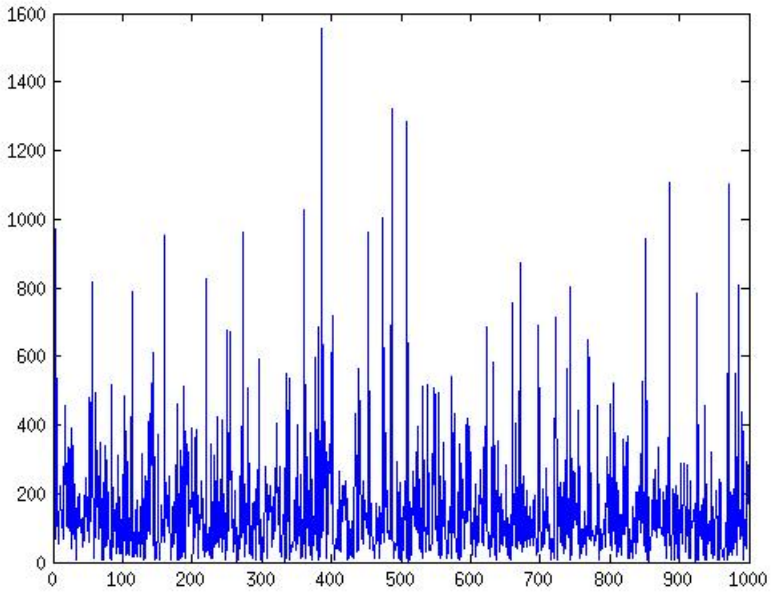

Figura 1: Búsqueda local iterada. En el eje y, el valor de la función objetivo en el punto actual.

En cambio, A12 realiza una búsqueda que es local, y solo se mueve si mejora el valor de la función objetivo. Por lo tanto, dicho valor será siempre decreciente.

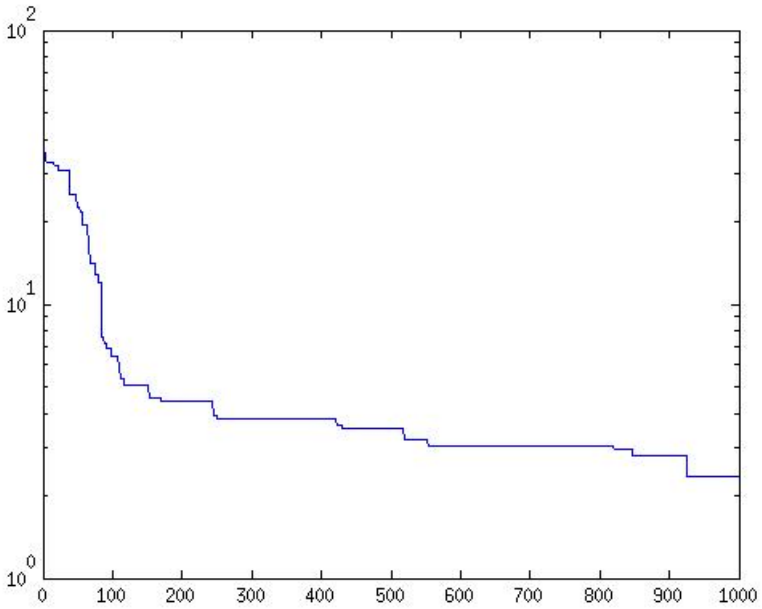

Figura 2: Búsqueda local. En el eje y, el valor de la función objetivo en el punto actual, en escala logarítmica.

Utilizamos escala logarítmica para apreciar mejor el decaimiento de la función objetivo. Notar que, al acercarnos al mínimo, las mejoras son cada vez más esporádicas, pues el radio de búsqueda se mantiene constante.

Veamos qué sucede con el algoritmo genético. En este caso, solo nos quedaremos con el valor de la función objetivo en el mejor de los individuos de la población actual.

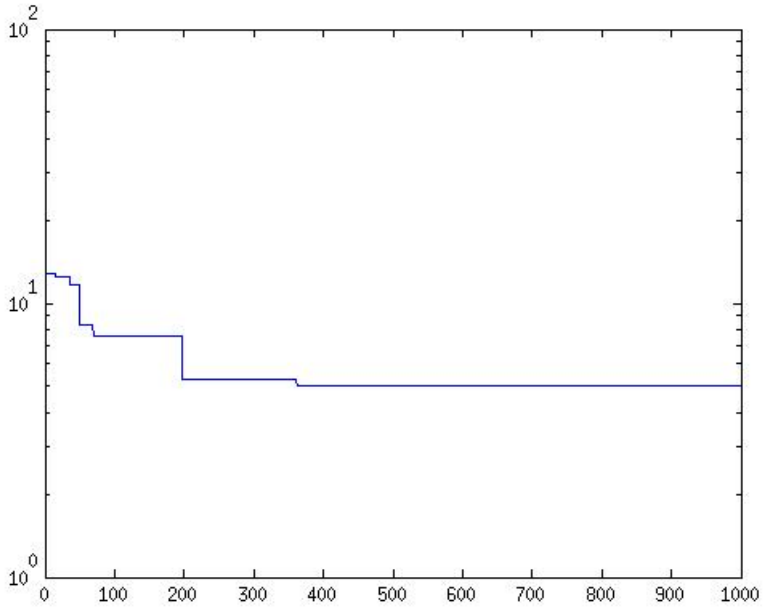

Figura 3: Algoritmo Genético. En el eje y, el valor de la función objetivo en el mejor elemento de la población actual, en escala logarítmica.

Resulta evidente algo que ya habíamos mencionado que podía ocurrir. El crossover y la mutación no resultaron suficientemente efectivos como para mejorar la población actual, a partir de cierto punto. Probamos además, incrementar la cantidad de iteraciones, pero las mejoras fueron cada vez menos significativas. Esto refuerza nuestra idea de que agregar mayor aleatoriedad en el armado de parejas puede ayudar a salir de estos puntos estancos, en los que los hijos no logran superar a ninguno de los padres.

Algo interesante de A3, recocido simulado, es que permite, con cierta probabilidad, elegir puntos que empeoran la solución. Para que se vea este hecho en acción, elegimos 1000 iteraciones, con una temperatura inicial alta (1000).

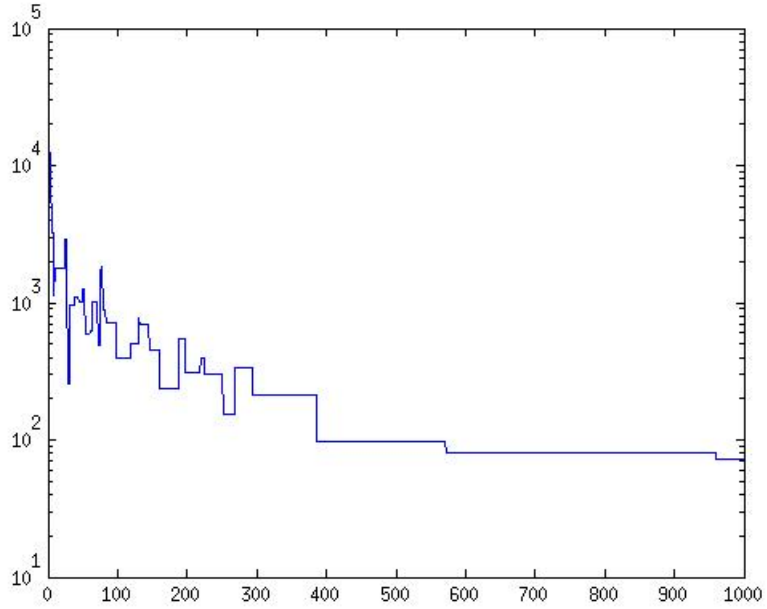

Figura 4: Recocido simulado. En el eje y, el valor de la función objetivo en el punto actual, en escala logarítmica.

Se ve claramente la oscilación del valor de la función objetivo, que es algo esperado en recocido simulado.

Para nuestra variante de recocido, agregamos reinicios, y una búsqueda local al final. Observamos que, según la elección del punto inicial, la cantidad de reinicios es distinta, pero en todos los casos obteniendo el mínimo global

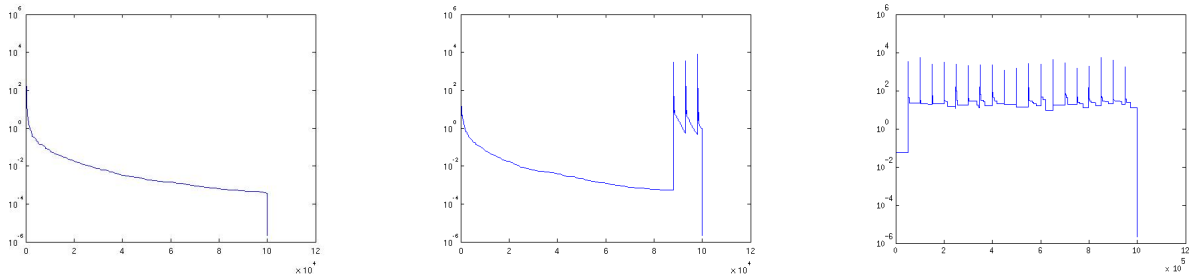

Figura 5: Versión nuestra de Recocido Simulado. Las imágenes se pueden ver en detalle en el archivo comprimido entregado.

Observamos varios comportamientos distintos, aunque todos tienen varias cosas en común. Después de un reinicio, la mejora es evidente y rápida, en general llegando a resultados similares. En varios casos, que no pudimos replicar para graficarlos, el primer intento (es decir, antes del primer reinicio) llevó al punto (-1,1,1,1,1,1,1), siendo el reinicio de gran ayuda.

Por otro lado, la búsqueda local final resulta en una mejora realmente importante, reduciendo en varios órdenes de magnitud el error, sin por ello afectar enormemente el tiempo de cómputo.

Podemos remarcar, entonces, dos tipos de ventajas que tiene nuestra propuesta. En primer lugar, usando los reinicios, evita que se caiga en un mínimo local indefinidamente. Se buscan localidades o vecindades con valor objetivo pequeño. En segundo lugar, la minimización local posterior, utilizando el método del gradiente, permite encontrar, dentro de la vecindad más prometedora encontrada, el mínimo local efectivo, esperando que realmente sea global.

Pudimos corroborar su funcionamiento en funciones particularmente patológicas. En Rosenbrock de dimensión 15 por ejemplo, en menos de diez segundos encuentra el mínimo global, con una precisión de 10^-6.

También probamos con otras funciones, con Bird (otra función muy oscilante, con muchos mínimos locales), también obtiene un rendimiento muy bueno.

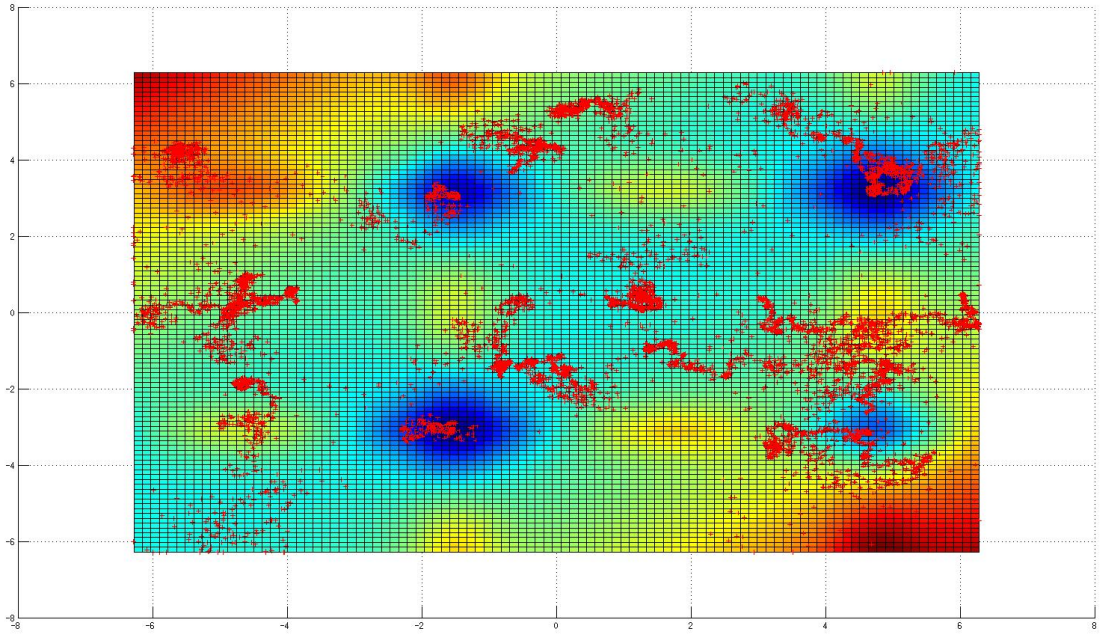

Figura 6: Versión mejorada de recocido simulado. Función Bird, en dos dimensiones. En escala de temperatura, el valor de la función en cada punto del rectángulo. En rojo, los puntos visitados por nuestro algoritmo (10000 total).

Podemos ver como, aunque no recorre todo el dominio de la función, su rengo es bastante amplio. Recorre lugares como la cercanía del (-5,5), con valores muy malos, pero al reiniciar logra salir de ahí. Por otro lado, los cuatro mínimos locales más pronunciados son visitados, y en todos los casos se adentra con la suficiente profundidad como para efectivamente llegar al mínimo local.

Sabemos que minimización global es un problema complejo, y que las heurísticas nos ayudan a llegar a una solución razonable, aunque tienen probabilidad de fallar. Un análisis de las características de una función pueden ayudar a determinar mejor los parámetros que minimizan esa probabilidad.